A Hitchhiker's Guide to the BIOS

Introduction

Remember, DON'T PANIC. This is the new, improved introduction to the Hitchhiker's Guide to the BIOS, which describes the BIOS (and many other aspects) of Atari's ST computer series. The introduction still won't tell you much, but at least it tells you not to panic.

The Guide's intended audience:

> Application writers (who will find some of the functions and hints here invaluable);
>
> Those wishing to make use of some of the ST's hardware-specific features (hacking palette colors, configuring the RS232 port, and so on);
>
> Those writing device drivers, video games, or cartridge-based applications;
>
> The habitually curious (including trivia trippers, information junkies, and documentation addicts).

For many reasons this should still be considered a preliminary document. A whole host of things remain undocumented, many GEMDOS issues have not even been approached by our friends at Digital Research, and there are a /whole lot/ of features we'd like to add to the software.

Periodically, as our roving reporters discover new ways to enjoy life on a roving reporter's budget of one Denebian slime dollar a day, we will be updating the Hitchhiker's Guide to reflect sudden, violent changes in reality. Those fortunates who do NOT own a Sub-etha Net auto-regressive pan-galactic update droid (if you DO own one, you know how difficult they are to get rid of) will have to call Atari occasionally to see if an update has occurred. We have no plans for another release before the end of September. Don't call /too/ often; there's an entire /galaxy/ of intelligent beings out there, and our operators are getting freaked out.

REWARD:
One Denebian Slime Dollar to the first discoverer of a misdocumentation error. /Two/ slime dollars to the second discoverer, and so on ....

GEMDOS BIOS Calls; Description and Deviation from the GEMDOS Spec.

The ST BIOS, contrary to the GEMDOS specification, is callable from the 68000's user mode.

The BIOS is re-entrant to three levels. That is, there may be up to three recursive BIOS calls before the system runs into trouble. No level checking is performed; the first sign of an overflow will be mysterious system behavior, and an eventual crash.

Applications should NOT attempt disk or printer I/O (this includes getbpb calls, and standard-output redirected to the printer device) in critical-error, system-timer or process-terminate handlers.

NOTE

> The BIOS modifies the function number (and the return address) pushed on the stack by the application. The function number on the stack will be ZERO on return. [For the curious: this feature saved several cycles per BIOS call ...]

```
 (0) getmpb
     VOID getmpb(p_mpb)
     LONG p_mpb;
```

Upon entry, 'p_mpb' points to a 'sizeof(MPB)' block to be filled in with the system initial Memory Parameter Block. Upon return, the MPB is filled in.

Structures are:

```
#define MPB     struct mpb
#define MD      struct md
```

```
#define PD      struct pd

MPB {
    MD *mp_mfl;             /* memory free list */
    MD *mp_mal;             /* memory allocated list */
    MD *mp_rover;           /* roving ptr */
};

MD {
    MD *m_link;             /* next MD (or NULL) */
    long m_start;           /* saddr of block */
    long m_length;          /* #bytes in block */
    PD *m_own;              /* owner's process descriptor */
};
```

[See `System Variables' for more information about setting up the initial TPA.]

(1) bconstat

```
WORD bconstat(dev)
WORD dev;
```

Return character-device input status, D0.L will be $0000 if no characters available, or $ffff if (at least one) character is available. 'dev' can be one of:

0	PRT: (printer, the parallel port)
1	AUX: (aux device, the RS232 port)
2	CON: (console, the screen)
3	MIDI port (Atari extension)
4	Keyboard port (Atari extension)
5	Raw console output

Legal operations on character devices are:

	(0)	(1)	(2)	(3)	(4)	(5)
Operation	PRT	AUX	CON	MIDI	KBD	RAW
bconstat()	no	yes	yes	yes	no	no
bconin()	yes	yes	yes	yes	no	no
bconout()	yes	yes	yes	yes	yes	yes
bcostat()	yes	yes	yes	yes	yes	no

The MIDI device has an interrupt-driven input buffer of 80 characters.

The keyboard device (#4) is output-only, and can be used to configure the intelligent keyboard (or drive it insane).

The raw console device (#5) prints characters to the screen without interpretation (control characters and escape sequences have no special meaning).

(2) bconin

```
WORD bconin(dev)
WORD dev;
```

'dev' is the character device number described in function 1.

Does not return until a character has been input (busy-wait). It returns the character value in D0.L, with the high word zero.

For the console (CON:, device 2) it returns the IBM-PC compatible scancode in the low byte of the upper word, and the Ascii character in the low byte of the low word.

If bit 3 in the system variable 'conterm' is set, then the high byte of the upper word will contain the value of the system variable 'kbshift' for that keystroke. [The default state for 'conterm%%3' is OFF.]

(3) bconout

```
WORD bconout(dev, c)
WORD dev, c;
```

'dev' is the character device number described in function 1.

Output character 'c' to the device. Does not return until the character has been written.

For PRT: returns 0 for failure and !0 for success.

(4) rwabs

```
LONG rwabs(rwflag, buf, count, recno, dev)
WORD rwflag;
LONG buf;
WORD count, recno, dev;
```

Read or write logical sectors on a device. 'rwflag' is one of:

0	read
1	write
2	read, do not affect media-change
3	write, do not affect media-change

'buf' points to a buffer to read or write to (unaligned transfers -- on odd boundaries -- are permitted, but they are slow). 'count' is the number of sectors to transfer. 'recno' is the logical sector number to start the transfer at. 'dev' is the device number, and on the ST is one of:

```
0   Floppy drive A:
1   Floppy drive B: (or "logical" drive A:
    on single-disk systems).
2+  Hard disks, networks, etc.
```

On return, 0L indicates a successful operation. Any negative number indicates an error condition. (It is the responsibility of the BIOS to detect media changes, and return the appropriate error code).

Modes 2 and 3 force a physical disk operation that will NOT affect media change, nor result in one (this allows the GEMDOS disk formatter, for instance, to read and write logical sectors after formatting a disk, and still allow the BIOS to recognize a media change on the volume just formatted).

[explain about "insert-disk" critical error hack for single-drive systems]

(5) setexc

```
LONG setexc(vecnum, vec)
WORD vecnum;
LONG vec;
```

'vecnum' is the number of the vector to get or set. 'vec' is the address to setup in the vector slot; no set is done if 'vec' is -1L. The vector's previous value is returned.

Vectors $00 through $FF are reserved for the 68000.

Logical vectors $100 through $1FF are reserved for GEMDOS. Vectors currently implemented are:

```
   $100  System timer interrupt
   $101  Critical error handler
   $102  Process terminate hook
$103..$107:  Currently unused, reserved
```

Logical vectors $200 through $FFFF are reserved for OEM use. The ST BIOS makes no provision for these.

(6) tickcal

```
LONG tickcal()
```

Returns a system-timer calibration value, to the nearest millisecond.

This is a silly function, since the number of elapsed milliseconds is passed on the stack during a system-timer trap.

(7) *getbpb

```
BPB *getbpb(dev)
WORD dev;
```

'dev' is a device number (0 for drive A, etc.) Returns a pointer to the BIOS Parameter Block for the specified drive, or 0L if (for some reason) the BPB cannot be determined.

(8) bcostat

```
LONG bcostat(dev)
```

'dev' is a character device number, as in function 1. Returns character output status:

-1	Device is ready to send (no waiting on next device-output call).
0	Device is not ready to send.

Note: Device 3 is keyboard and 4 is midi.

(9) mediach

```
LONG mediach(dev)
WORD dev;
```

'dev' is a drive number. Returns one of:

0	Media definitely has not changed
1	Media /might/ have changed
2	Media definitely has changed

GEMDOS will respond to a return value of '1' with a read operation. If the BIOS detects an absolute media change, it will return a "media change" error at that time.

(10) drvmap

```
LONG drvmap()
```

Returns a bit-vector that contains a '1' in a bit position (0 .. 31) when a drive is available for that bit, or a 0 if there is no drive available for the bit.

Installable disk drivers must correctly maintain the longword '_drvbits' [see: System Variables].

(11) kbshift

```
LONG kbshift(mode)
WORD mode;
```

If 'mode' is non-negative, sets the keyboard shift bits accordingly and returns the old shift bits. If 'mode' is less than zero, returns the IBM-PC compatible state of the shift keys on the keyboard, as a bit-vector in the low byte of D0.

Bit assignments are:

0	Right shift key
1	Left shift key
2	Control key
3	ALT key
4	Caps-lock
5	Right mouse button (CLR/HOME)
6	Left mouse button (INSERT)
7	(reserved, currently zero)

Extended BIOS Functions

These functions are available through trap 14. The calling conventions are the same as for trap 13. Contrary to the GEMDOS specification, the caller does NOT have to be in supervisor mode. It is the caller's responsibility to cleanup arguments passed to the trap (as per the C calling standard).

A typical trap handler, one that works from a C binding, might be:

```
_trap14:
        move.l  (sp)+,tr14ret       ; pop ret addr
        trap    #14                 ; do BIOS func
        move.l  tr14ret,-(sp)       ; return to
        rts                         ;  caller

        bss
tr14ret: ds.l                       ; saved ret. addr
```

and it might be used like:

```
/*
 * Stupid way to set the screen to a single value.
 */
set_screen_to(v)
WORD v;
{
    extern long trap14();
    register WORD *p;
    register int i;

    scrbase = (WORD *)trap14(3);
    for (i = 0x4000; i; --i)
        *p++ = v;
}

/*
 * Xor palettes in a range with a given value
 */
set_palette_range(start, fin, v)
WORD start, fin, v;
{
    while (start <= fin)
        trap14(7, trap14(7, -1) ^ v);
    }
```

```
(0) initmous
    VOID initmous(type, param, vec)
    WORD type;
    LONG param, vec;
```

Initialize mouse packet handler. 'type' is one of:

type	Action
0	disable mouse
1	enable mouse, in relative mode
2	enable mouse, in absolute mode
3	(unused)
4	enable mouse, in keycode mode

'param' points to a parameter block that should look like:

```
struct param {
    BYTE topmode;
    BYTE buttons;
    BYTE xparam;
    BYTE yparam;
};
```

'topmode' should be:

0	Y_position == 0 at bottom
1	Y_position == 0 at top

'buttons' is a parameter for the keyboard's "set mouse buttons" command.

'xparam' and 'yparam' are the X and Y threshold, scale or delta factors, depending on the mode the mouse is being placed in.

For mouse absolute mode, some extra parameters immediately follow the parameter block:

```
struct extra {
    WORD xmax;
    WORD ymax;
    WORD xinitial;
    WORD yinitial;
};
```

'xmax' and 'ymax' specify the maximum X and Y mouse positions. 'xinitial' and 'yinitial' specify the initial X and Y mouse position.

'vec' points to a mouse interrupt handler; see extended function number 34, 'kbdvbase', for further information about ikbd subsystem handlers.

(1) ssbrk

```
LONG ssbrk(amount)
WORD amount;
```

Reserve 'amount' bytes from the top of memory. Returns a long pointing to the base of the allocated memory. This function MUST be called before the OS is initialized.

'ssbrk' is actually pretty useless. It DOES NOT work after GEMDOS has been brought up, since the TPA has already been set up.

(2) _physBase

```
LONG _physBase()
```

Get the screen's physical base address (at the beginning of the next vblank).

(3) _logBase

```
LONG _logBase()
```

Get the screen's logical base, right away. This is the location that GSX uses when drawing to the screen.

(4) _getRez

```
WORD _getRez()
```

Get the screen's current resolution (returning 0, 1 or 2).

(5) _setScreen

```
VOID _setScreen(logLoc, physLoc, rez)
LONG logLoc, physLoc;
WORD rez;
```

Set the logical screen location (logLoc), the physical screen location (physLoc), and the physical screen resolution. Negative parameters are ignored (making it possible, for instance, to set screen resolution without changing anything else).

The logical and physical screen locations change immediately.

When resolution is changed, the screen is cleared, the cursor is homed, and the VT52 terminal emulator state is reset.

(6) _setPallete
```
VOID _setPallete(palettePtr)
LONG palettePtr;
```
Set the contents of the hardware palette register (all 16 color entries) from the 16 words pointed to by 'palettePtr'. 'paletteptr' MUST be on a word boundary. The palette assignment takes place at the beginning of the next vertical blank interrupt.

(7) _setColor
```
WORD _setColor(colorNum, color)
WORD colorNum, color;
```
Set the palette number 'colorNum' in the hardware palette table to the given color. Return the old color in D0.W. If 'color' is negative, the hardware register is not changed.

(8) _floprd
```
WORD _floprd(buf, filler, devno, sectno, trackno,
    sideno, count)
LONG buf, filler;
WORD devno, sectno, trackno, sideno, count;
```
Read one or more sectors from a floppy disk. 'filler' is an unused longword. 'buf' must point to a word-aligned buffer large enough to contain the number of sectors requested. 'devno' is the floppy number (0 or 1). 'sectno' is the sector number to start reading from (usually 1 through 9). 'trackno' is the track number to seek to. 'sideno' is the side number to select. 'count' is the number of sectors to read (which must be less than or equal to the number of sectors per track).

On return, D0 contains a status code. If D0 is zero, the operation succeeded. If D0 is nonzero, the operation failed (and D0 contains an error number).

(9) _flopwr

```
WORD _flopwr(buf, filler, devno, sectno, trackno,
     sideno, count)
LONG buf, filler;
WORD devno, sectno, trackno, sideno, count;
```

Write one or more sectors to a floppy disk. 'buf' must point to a word-aligned buffer. 'filler' is an unused longword. 'devno' is the floppy number (0 or 1). 'sectno' is the sector number to start writing to (usually 1 through 9). 'trackno' is the track number to seek to. 'sideno' is the side number to select. 'count' is the number of sectors to write (which must be less than or equal to the number of sectors per track).

On return, D0 contains a status code. If D0 is zero, the operation succeeded. If D0 is nonzero, the operation failed (and D0 contains an error number).

Writing to the boot sector (sector 1, side 0, track 0) will cause the media to enter the "might have changed" state. This will be reflected on the next rwabs() or mediach() BIOS call.

(10) _flopfmt

```
WORD _flopfmt(buf, filler, devno, spt, trackno, sideno,
     interlv, magic, virgin)
LONG buf, filler;
WORD devno, spt, trackno, sideno, interlv, virgin;
LONG magic;
```

Format a track on a floppy disk. 'buf' must point to a word-aligned buffer large enough to hold an entire track image (8K for 9 sectors-per-track). 'filler' is an unused longword. 'devno' is the floppy drive number (0 or 1). 'spt' is the number of sectors-per-track to format (usually 9). 'trackno' is the track number to format (usually 0 to 79). 'sideno' is the side number to format (0 or 1). 'interlv' is the sector-interleave factor (usually 1). 'magic' is a magic number that MUST be the value $87654321. 'virgin' is a word fill value for new sectors.

On return, D0 contains a status code. If D0 is zero, the operation succeeded. If D0 is nonzero, the operation failed (and D0 contains an error number). The format function can soft-fail when it finds bad sectors during the verify pass. The caller has the choice of attempting to re-format the media, or recording the bad sectors so they will not be included in the file system.

A null-terminated (0.W) list of bad sector numbers is returned in the buffer. They are not necessarily in numerical order. (If there were no bad sectors, the first word in the buffer will be zero.)

A good value for 'virgin' is $E5E5. The high nibble of each byte in the 'virgin' parameter must not be equal to $F. Resist the temptation to format a disk with sectors initialized to zero.

Formatting a track will cause the media to enter the "definitely changed" state. This will be reflected on the next rwabs() or mediach() BIOS call.

```
(11) used-by-BIOS
    VOID used-by-BIOS()
          [Obsolete function]
```

```
(12) midiws
    VOID midiws(cnt, ptr)
    WORD cnt;
    LONG ptr;
```

Writes a string to the MIDI port. 'cnt' is the number of characters to write, minus one. 'ptr' points to a vector of characters to write.

```
(13) _mfpint
    VOID _mfpint(interno, vector)
    WORD interno;
    LONG vector;
```

Set the MFP interrupt number 'interno' (0 to 15) to 'vector'. The old vector is written over (and thus unrecoverable).

```
(14) iorec
    LONG iorec(devno)
    WORD devno;
```

Returns a pointer to a serial device's input buffer record. 'devno' is one of:

devno	Device
0	RS232
1	Keyboard
2	MIDI

The structure of the record is:

```
struct iorec
{
    LONG ibuf;          /* pointer to buffer */
    WORD ibufsiz;       /* size of buffer */
    WORD ibufhd;        /* head index */
    WORD ibuftl;        /* tail index */
    WORD ibuflow;       /* low-water mark */
    WORD ibufhi;        /* high-water mark */
};
```

For RS-232, an output-buffer record immediately follows the input-buffer record. The format of the output-buffer record is identical.

'ibuf' points to the device's buffer. 'ibufsiz' is the buffer's size. 'ibufhi' is the buffer's high-water mark. 'ibuflow' is the buffer's low-water mark.

If flow control is enabled and the number of characters in the buffer reaches the high-water mark, the ST requests (according to the flow-control protocol) the sender to stop sending characters. When the number of characters in the buffer drops below the low-water mark, the ST tells the sender to resume transmission.

The flow-control operation is similar for the RS-232 output record.

(15) rsconf

```
LONG rsconf(speed, flowctl, ucr, rsr, tsr, scr)
WORD speed, flowctl, ucr, rsr, tsr, scr;
```

Configure RS-232 port. If any parameter is -1 ($FFFF), the corresponding hardware register is not set. 'speed' sets the port's baud rate, as per:

speed	Rate (bps)
0	19,200
1	9600
2	4800
3	3600
4	2400
5	2000
6	1800
7	1200
8	600
9	300

10	200
11	150
12	134
13	110
14	75
15	50

'flow' sets the flow control, as per:

flow	Flavor
0	No flow control [powerup default]
1	XON/XOFF (^S/^Q)
2	RTS/CTS
3	XON/XOFF and RTS/CTS [is this useful?]

'ucr', 'rsr', 'tsr', and 'scr' set the appropriate 68901 registers.

Returns old values of ucr, rsr, tsr, and scr (that order) byte packed in a long value.

(16) keytbl

```
    LONG keytbl(unshift, shift, capslock)
    LONG unshift, shift, capslock;
```

Sets pointers to the keyboard translation tables for unshifted keys, shifted keys, and keys in caps-lock mode. Returns a pointer to the beginning of a structure:

```
    struct keytab {
        LONG unshift;           /* -> unshift table */
        LONG shift;             /* -> shift table */
        LONG capslock;          /* -> capslock table */
    };
```

Each pointer in the structure should point to a table 128 bytes in length. A scancode is converted to Ascii by indexing into the table and taking the byte there.

(17) _random

```
    LONG _random()
```

Returns a 24-bit psuedo-random number in D0.L. Bits 24..31 will be zero. The sequence /should/ be different each time the system is turned on. [The algorithm is from vol. 2 of Knuth:

$$S = [S * C] + K$$

where K = 1, C = 3141592621, and S is the seed. S >> 8 is returned. The initial value of S is taken from the frame-counter '_frclock'.]

The function's behavior is surprisingly good, except that bit 0 has an /exact/ distribution of 50%. Therefore it is probably not a good idea to test individual bits and expect them to be well behaved.

(18) _protobt

```
VOID _protobt(buf, serialno, disktype, execflag)
LONG buf, serialno;
WORD disktype, execflag;
```

Prototype an image of a boot sector. Once the boot sector image has been constructed with this function, write it to the volume's boot sector.

'buf' points to a 512-byte buffer (which may contain garbage, or already contain a boot sector image).

'serialno' is a serial number to stamp into the boot sector. If 'serialno' is -1, the boot sector's serial number is not changed. If 'serialno' is greater than or equal to $01000000, a random serial number is generated and placed in the boot sector.

'disktype' is either -1 (to leave the disk type information alone) or one of the following:

```
0: 40 tracks, single sided (180K)
1: 40 tracks, double sided (360K)
2: 80 tracks, single sided (360K)
3: 80 tracks, double sided (720K)
```

If 'execflag' is 1, the boot sector is made executable. If 'execflag' is 0, the boot sector is made non-executable. If 'execflag' is -1, the boot sector remains executable or non-executable depending on the way it was originally.

(19) _flopver

```
WORD _flopver(buf, filler, devno, sectno, trackno,
     sideno, count)
LONG buf, filler;
WORD devno, sectno, trackno, sideno, count;
```

Verify (by simply reading) one or more sectors from a floppy disk. 'buf' must point to a word-aligned 1024-byte buffer. 'filler' is an unused longword. 'devno' is the floppy number (0 or 1). 'sectno' is the sector number to start reading from (usually 1 through 9). 'trackno' is the track number to seek to. 'sideno' is the side number to select. 'count' is the number of sectors to verify (which must be less than or equal to the number of sectors per track).

On return, D0 contains a status code. If D0 is zero, the operation succeeded. If D0 is nonzero, the operation failed (and D0 contains an error number).

A null-terminated (0.W) list of bad sector numbers is returned in the buffer. They are not necessarily in numerical order. (If there were no bad sectors, the first word in the buffer will be zero.)

(20) scrdmp

```
VOID scrdmp()
```

Dump screen to printer. [Currently this is the monochrome-only version from CES. Will be fixed soon.]

(21) cursconf

```
WORD cursconf(function, operand)
WORD function, operand;
```

Configure the "glass terminal" cursor. The 'function' code is one of the following:

0	Hide cursor
1	Show cursor
2	Cursor set to blink
3	Cursor set not to blink
4	Set cursor blink timer to 'operand'
5	Return cursor blink timer value

The cursor blink rate is based on the video scan rate (60hz for color, 70hz for monochrome, 50hz for PAL). The 'rate' parameter is equal to one-half the cycle time.

(22) settime

```
VOID settime(datetime)
LONG datetime;
```

Sets the intelligent keyboard's idea of the time and date. 'datetime' is a 32-bit DOS-format date and time (time in the low word, date in the high word).

(23) gettime

```
LONG gettime()
```

Interrogates the intelligent keyboard's idea of the time and date, and returns that value (in DOS format) as a 32-bit word. (Time in the low word, date in the high word).

(24) bioskeys

```
VOID bioskeys()
```

Restores the powerup settings of the keyboard translation tables.

(25) ikbdws

```
VOID ikbdws(cnt, ptr)
WORD cnt;
LONG ptr;
```

Writes a string to the intelligent keyboard. 'cnt' is the number of characters to write, minus one. 'ptr' points to a vector of characters to write.

(26) jdisint

```
VOID jdisint(intno)
WORD intno;
```

Disable interrupt number 'intno' on the 68901.

(27) jenabint

```
VOID jenabint(intno)
WORD intno;
```

Enable interrupt number 'intno' on the 68901.

(28) giaccess

```
BYTE giaccess(data, regno)
BYTE data;
WORD regno;
```

Read or write a register on the sound chip. 'regno' is the register number, logically ORed with:

```
$00 to read [well, ok, you don't
    /really/ OR with this...]
$80 to write
```

'data' is a byte to write to the register.

Sound chip registers are not shadowed. Procedures that change register values by reading a register, modifying a local copy of it, and writing the result back to the register, should be critical sections. In particular, the BIOS (frequently) updates the PORT A register, and any code that read-modify-writes PORT A must be atomic. [See GIACCESS at the end of this guide]

(29) offgibit

```
VOID offgibit(bitno)
WORD bitno;
```

Atomically set a bit in the PORT A register to zero.

(30) ongibit

```
VOID ongibit(bitno)
WORD bitno;
```

Atomically set a bit in the PORT A register to one.

(31) xbtimer

```
VOID xbtimer(timer, control, data, vec)
WORD timer, control, data;
LONG vec;
```

'timer' is the timer number (0, 1, 2, 3 corresponding to 68901 timers A, B, C and D). 'control' is the timer's control-register setting. 'data' is a byte shoved into the timer's data register. 'vec' is a pointer to an interrupt handler.

Timers are allocated:

Timer	Usage
A	Reserved for end-users and applications
B	Reserved for graphics (hblank sync, etc.)
C	System timer (200hz)
D	RS-232 baud-rate control (this timer's interrupt vector is available to anyone).

(32) dosound

```
VOID dosound(ptr)
LONG ptr;
```

Set sound daemon's "program counter" to 'ptr'. 'ptr' points to a set of commands organized as bytes.

Command numbers $00 through $0F take a one byte argument to be shoved into a sound chip register. (Command $00 shoves the byte into register 0, command 1 shoves the byte into register 1, and you get the idea...)

Command $80 takes a one byte argument which is shoved into a temporary register.

Command $81 takes three one-byte arguments. The first argument is a register number to load, using the temp register. The second argument is a 2's complement value to be added to the temp register. The third argument is the termination value. The instruction is executed (once on each update) until the temp register equals the termination value.

Commands $82 through $FF take a one-byte argument. If the argument is zero, the sound is terminated. Otherwise the argument reflects the number of system-timer ticks (at 50hz) until the next update.

(33) setprt

```
WORD setprt(config)
WORD config;
```

Set/get printer configuration byte. If 'config' is -1 ($FFFF) return the current printer configuration byte. Otherwise set the byte and return it's old value.

Bits currently defined are:

Bit#	When 0	When 1
0	Dot matrix	Daisy wheel
1	Color device	Monochrome device
2	Atari printer	"Epson" printer
3	Draft mode	Final mode
4	Parallel port	RS232 port
5	Form-feed	Single sheet
6	reserved	
7	reserved	
8	reserved	
9	reserved	
10	reserved	
11	reserved	
12	reserved	
13	reserved	
14	reserved	
15	Must be zero	

(34) kbdvbase

```
LONG kbdvbase()
```

Returns a pointer to the base of a structure:

```
struct kbdvecs {
    LONG midivec;       /* MIDI-input */
    LONG vkbderr;       /* keyboard error */
    LONG vmiderr;       /* MIDI error */
    LONG statvec;       /* ikbd status packet */
    LONG mousevec;      /* mouse packet */
    LONG clockvec;      /* clock packet */
    LONG joyvec;        /* joystick packet */
    LONG midisys;       /* system MIDI vector */
    LONG ikbdsys;       /* system IKBD vector */
};
```

'midivec' is initialized to point to a buffering routine in the BIOS. D0.B will contain a character read from the MIDI port.

'vkbderr' and 'vmiderr' are called whenever an overrun condition is detected on the keyboard or MIDI 6850s. [Probably not a useful vector to grab.]

'statvec', 'mousevec', 'clockvec', and 'joyvec' point to ikbd status, mouse, real-time clock, and joystick packet handlers. The packet handlers are passed a pointer to the packet received in A0, and on the stack as a LONG. GEM/GSX uses the mouse vector. Handlers should return with an RTS, and should not spend more than 1ms handling the interrupt.

The 'midisys' and 'ikbdsys' vectors are called when characters are available on the appropriate 6850. Initially they point to default routines (the MIDI handler indirects through 'midivec', and the ikbd handler parses-out ikbd packets and calls the appropriate subsystem vectors).

(35) kbrate

```
WORD kbrate(initial, repeat)
WORD initial, repeat;
```

Get/set the keyboard's repeat rate. 'initial' governs the initial delay (before key-repeat starts). 'repeat' governs the rate at which key-repeats are generated. If a parameter is -1 ($FFFF) it is not changed. Times are based on system ticks (50hz).

Returns the old key-repeat values, with 'initial' in the high byte of the low word and 'repeat' in the low byte of the low word.

(36) _prtblk

```
VOID _prtblk()
```

Prtblk() primitive [see manual pages on PRTBLK].

(37) vsync

```
VOID vsync()
```

Waits until the next vertical-blank interrupt and returns. Useful for synchronizing graphics operations with vblank.

(38) supexec

```
VOID supexec(codeptr)
LONG codeptr;
```

'codeptr' points to a piece of code, ending in an RTS, that is executed in supervisor mode. The code cannot perform BIOS or GEMDOS calls. This function is meant to allow programs to hack hardware and protected locations without having to fiddle with GEMDOS get/set supervisor mode call.

(39) puntaes

VOID puntaes()

Throws away the AES, freeing up any memory it used. If the AES is still resident, it will be discarded and the system will reboot. If the AES is not resident (if it was discarded earlier) the function will return.

There is NO way to throw away the AES and return -- the reboot MUST be performed. [Ok, ok -- we know this is a lose.]

CONOUT Escape Sequences

These are the escape functions interpreted by the BIOS′ conout() function. For the most part they emulate a VT-52 terminal [that′s the easy one to do]. There are extensions to hack screen colors, control screen wrap, and a few other simple functions.

ESC A
Cursor Up
This sequence moves the cursor up one line. If the cursor is already on the top line of the screen, this sequence has no effect.

ESC B
Cursor Down
This moves the cursor down one line. If the cursor is already on the last line of the screen, this escape sequence has no effect.

ESC C
Cursor Forward
This moves the cursor one position to the right. If this function would move the cursor off the screen, this sequence has no effect.

ESC D
Cursor Backward
This move the cursor one position to the left. This is a non- destructive move because the character over which the cursor now rests is not replaced by a blank. If the cursor is already in column 0, this escape sequence has no effect.

ESC E
Clear Screen (and Home Cursor)
This moves the cursor to column 0, row I (the top left-hand corner of the screen), and clears all characters from the screen.

ESC H
Home Cursor
This move the cursor to column 0, row 0. The screen is NOT cleared.

ESC I
Reverse Index
Moves the cursor to the same horizontal position on the

preceding lines. If the cursor is on the top line, a scroll down is performed.

ESC J
Erase to End of Page
Erases all the information from cursor (including cursor position) to the end of the page.

ESC K
Clear to End of Line
This sequence clears the line from the current cursor position to the end of the line.

ESC L
Insert Line
Inserts a new blank line by moving the line that cursor is on, end all following lines, down one line. Then, the cursor is moved to the beginning of the new blank line.

ESC M
Delete Line
Deletes the contents of the line that the cursor is on, places the cursor at the beginning of the line, moves all the following lines up one line, and adds a blank line at the bottom.

ESC Y
Position Cursor
The two characters that follow the "Y" specify the row and column to which the cursor is to be moved. The first character specifies the row, the second specifies the colum. Rows and columns number from 1 up.

ESC b
Set Foreground Color
The Foreground Color is the color in which the character is displayed.

Escape-b must be followed by a color selection character. Only the four least significant bits of the color character are used:

Bit Pattern of Control Byte:

```
   7     6     5     4     3     2     1     0
+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |                       |
|  X  |  X  |  X  |  X  |      color index      |
|     |     |     |     |                       |
+-----+-----+-----+-----+-----+-----+-----+-----+
(X = "don't care")
```

ESC c
Set Background Color

This function selects Background Color, the color of the cell that contains the characters.

Escape-c must be followed by a color selection character. Only the four least significant bits of the color character are used. (See diagram for ESC-b function)

ESC d
Erase Beginning of Display

This sequence erases from beginning of the display to the cursor position. The cursor position is erased also.

ESC e
Enable Cursor

This sequence causes the cursor to be invisible. The cursor may still be moved about on the display, using escape sequence defined in this appendix.

ESC f
Disable Cursor

This sequence causes the cursor to be invisible. The cursor may still be moved about on the display, using escape sequences defined in this appendix.

ESC j
Save Cursor Position

This sequence preserves the current cursor position. You can restore the cursor to the previously saved position with ESC-k.

ESC k
Restore Cursor Position

This sequence restores the cursor to a previously saved position. If you use this sequence without having previously saved the cursor position, then the cursor is moved to the home position, the top left-hand corner of the screen.

ESC l
Erase Entire Line

This sequence erases an entire line and moves the cursor to the leftmost column.

ESC o
Erase Beginning of Line

Erases from the beginning of the line to the cursor and includes the cursor position.

ESC p
Enter Reverse Video Mode

Enters the reverse video mode so that characters are displayed as background color characters on a foreground colored cell.

ESC q
Exit Reverse Video Mode
Exits the reverse video mode.

ESC v
Wrap at End of Line
This sequence causes the first character past the last displayable position on a line to be automatically placed in the first character position on the next line. The page scrolls up if necessary.

ESC w
Discard at End of Line
Following invocation of this sequence, after the last displayable character on a line has been reached, the characters overprint. Therefore, only the last character received is displayed in the last column position.

Traps, Interrupts and Interrupt Vectors

The ST makes use of four of the sixteen TRAP vectors provided by the 68000. All other traps are available for applications.

Trap	Use
0	(none)
1	GEMDOS interface
2	DOS extensions (GEM, GSX)
3	(none)
4	(none)
5	(none)
6	(none)
7	(none)
8	(none)
9	(none)
10	(none)
11	(none)
12	(none)
13	BIOS
14	Atari BIOS extensions
15	(none)

68901 interrupts are based at $100. The sixteen longwords at this location are bound by the hardware to:

Vector	Function
$100	(disabled) Parallel port int.
$104	(disabled) RS232 Carrier Detect
$108	(disabled) RS232 Clear-To-Send
$10c	(disabled)
$110	(disabled)
$114	200hz System clock
$118	Keyboard/MIDI [6850]
$11c	(disabled) Polled FDC/HDC
$120	HSync (initially disabled)
$124	RS232 transmit error
$128	RS232 transmit buffer emtpy
$12c	RS232 receive error
$130	RS232 receive buffer full
$134	(disabled)
$138	(disabled) RS232 ring indicator
$13c	(disabled) Polled monitor type

The divide-by-zero vector is pointed at an RTE.

All other traps (Bus Error, et al) are pointed at a handler that dumps the processor state and attempts to terminate the current process. [See: System Initialization]

The Line 1010 ("Line Aye") vector is used as a short-circuit around the VDI to the ST's graphics primitives. It is a powerful and useful interface; see the `Line A' document for further information.

The Line 1111 ("Line Eff") trap is currently being used internally to the system. If you fiddle with this vector the AES will break.

The FDC/HDC interrupt may be enabled by a hard disk device driver. The floppy disk code, however, assumes this interrupt is disabled (it busy-waits on the input bit's state). It is the responsibility of other drivers in the system to ensure that, when the floppy disk read/write/format code gets control, the FDC/HDC interrupt is disabled.

The processor's normal interrupt priority level is 3. This is to prevent HBLANK (autovector level 2) interrupts from occurring on every scanline. [It would eat about 10% of a system running in a color graphics mode, or about 22% of a system running in monochrome. Yuck.] The default HBLANK interrupt handler modifies the interrupted process' IPL to 3 and performs an RTE. This is to discourage programs from using IPL 0 -- to use HBLANK, use an IPL of 1.

To prevent "jittering" in programs that change screen colors on the fly, using the HBLANK and HSYNC interrupt vectors, the following hack will keep the system intact and still yield a solid display:

[1] Re-vector the keyboard/MIDI interrupt to a routine that lowers the IPL to 5 and then jumps through the original vector.

[2] During the "critical" section of the screen, re-vector the 200hz system clock interrupt vector to point to a routine that increments a counter and RTEs. The counter keeps track of the number of system ticks that occur during the critical section.

[3] After the critical section, block interrupts (at IPL 6) and call the sytem clock handler (JMP through the interrupt vector, with a fake SR and return address on the stack) the number of times indicated by the counter.

Calling the BIOS From an Interrupt Handler

It is possible to do a BIOS call from an interrupt handler. More specifically, it is possible for EXACTLY ONE interrupt handler to call the BIOS at a time. It is NOT possible to do GEMDOS, VDI or AES traps from interrupt handlers.

The basic problem is a critical section in the BIOS trap handler code. The critical section occurs when the registers are being saved or restored in the register save area; the variable ``savptr'' must be maintained correctly.

```
*+
*  Calling the BIOS from an interrupt, safely.
*
*-

* These are from the BIOS listing:
savptr  =       $4a2    ; BIOS register-save ptr
sav_amt =       23*2    ; #words BIOS saves on the stack

interrupt_handler:
        .
        .

* Create safe TRAP environment:
        sub.l   #sav_amt,savptr
        .
        .
        . lotsa BIOS traps (#13, #14 only)
        .

* Restore old trap environment:
        add.l   #sav_amt,savptr
        .
        .
        rte                     ; (or whatever)
```

--- DANGER ---

Only ///ONE/// interrupt handler may do this. That is, two interrupt handlers cannot nest and do BIOS calls in this manner.

System Variables

This is a list of variables in the ST BIOS that have been "cast in concrete". Their locations and meanings in future revisions of the ST BIOS are guarenteed not to change.

Any other variables in RAM, routines in the ROM, or vectors below $400 that are not documented here are almost certain to change. It is important not to depend on undocumented variables or ROM locations.

etv_timer (long) $400
    Timer handoff vector (logical vector $100). See GEMDOS documentation.

etv_critic (long) $404
    Critical error handoff vector (logical vector $101). See GEMDOS documentation.

etv_term (long) $408
    Process-terminate handoff vector (logical vector $102). See GEMDOS documentation.

etv_xtra (longs) $40c
    Space for logical vectors $103 through $107).

memvalid (long) $420
    Contains the magic number $752019F3, which (together with 'memval2') validates 'memcntlr' and indicates a successful coldstart.

memcntlr (byte) $424
    Contains memory controller configuration nibble (the low nibble). For the full story, see the hardware manual.

resvalid (long) $426
    If 'resvalid' is the magic number $31415926 on system RESET, the system will jump though 'resvector'.

resvector (long) $42a
    System-RESET bailout vector, valid if 'resvalid' is a magic number. Called early-on in system initialization (before /any/ hardware registers, including the memory controller configuration register, have been touched). A return address will be loaded into A6. Both stack pointers will contain garbage. (See Rainbow TOS Release Notes.)

phystop (long) $42e
Physical top of RAM. Contains a pointer to the first unusable byte (i.e. $80000 on a 512K machine).

_membot (long) $432
Bottom of available memory. The 'getmpb' BIOS function uses this value as the start of the GEMDOS TPA.

_memtop (long) $436
Top of available memory. The 'getmpb' BIOS function uses this value as the end of the GEMDOS TPA.

memval2 (long) $43a
Contains the magic number $237698AA which (together with 'memvalid') validates 'memcntlr' and indicates a successful coldstart.

flock (word) $43e
Used to lock usage of the DMA chip. Should be nonzero to ensure that the OS does not touch the DMA chip registers during vertical blank. Device-driver writers TAKE NOTE: this variable MUST be nonzero in order to make use of the DMA bus.

seekrate (word) $440
Default floppy seek rate. Read only at boot time: setting this variable has no effect until you reboot. Bits zero and one contain the default floppy disk seek rate for both drives:

00	6ms
01	12ms
10	2ms
11	3ms [default]

_timr_ms (word) $442
System timer calibration (in ms). Should be $14 (20 decimal), since the timer handoff vector is called at 50hz. Returned by BIOS function '_tickcal', and passed on the stack to the timer handoff vector.

_fverify (word) $444
Floppy verify flag. When nonzero, all writes to floppies are read-verified. When zero, no write-verifies take place. The default state (after system-reset) is to verify.

_bootdev (word) $446
Contains the device number the system was booted from. The BIOS constructs an environment string from this variable before bringing up the desktop.

palmode (word) $448
When nonzero, indicates the system is in PAL (50hz video) mode. When zero, indicates the system is in NTSC (60hz video) mode.

defshiftmd (byte) $44a
Default video resolution. If the system is forced to change from monochrome mode to a color resolution, 'defshiftmd' contains the resolution the system will switch to.

sshiftmd (word) $44c
Contains shadow for 'shiftmd' hardware register:

0	320x200x4 (low resolution)
1	640x200x2 (medium rez)
2	640x400x1 (high rez / "monochrome")

_v_bas_ad (long) $44e
Pointer to base of screen memory. On a 512-byte boundary on ST and Mega, a 2-byte boundary on STE and an 8-byte boundary on TT. Always points to 32K of contiguous memory. This is "logbase."

vblsem (word) $452
Semaphore to enforce mutual exclusion in vertical-blank interrupt handler. Should be '1' to enable vblank processing,

nvbls (word) $454
Number of longwords that '_vblqueue' points to. (On RESET, defaults to 8).

_vblqueue (long) $456
Pointer to a vector of pointers to vblank handlers.

colorptr (long) $45a
Pointer to a vector of 16 words to load into the hardware palette registers on the next vblank. If NULL, the palettes are not loaded. 'colorptr' is zeroed after the palettes are loaded.

screenpt (long) $45e
Pointer to the base of screen memory, to be setup on the next vblank. If NULL, the screen base is not changed.

_vbclock (long) $462
Count of vertical-blank interrupts.

_frclock (long) $466
Count of vertical-blank interrupts that were processed (not blocked by 'vblsem').

hdv_init (long) $46a
Vector to hard disk initialization.

swv_vec (long) $46e
The system follows this vector when it detects a transition on the "monochrome monitor detect" input (from low to high rez, or visa-versa). 'swv_vec' initially points to the system reset handler; therefore the system will reset if the user changes monitors.

hdv_bpb (long) $472
Vector to routine to return a hard disk's Bios Parameter Block (BPB). Same calling conventions as the BIOS function for GETBPB.

hdv_rw (long) $476
Vector to routine to read or write on a hard disk. Same calling conventions as the BIOS function for RWABS.

hdv_boot (long) $47a
Vector to routine to boot from hard disk.

hdv_mediach (long) $47e
Vector to routine to return a hard disk's media change mode. Same as BIOS binding for floppies.

_cmdload (word) $482
When nonzero an attempt is made to load and execute COMMAND.PRG from the boot device. (Load a shell or an application in place of the desktop). Can be set to nonzero by a boot sector.

conterm (byte) $484
Contains attribute bits for the console system:

Bit	Function	
0	nonzero:	enable key-click
1	nonzero:	enable key-repeat
2	nonzero:	enable bell when ^G is written to CON:
3	nonzero:	on BIOS conin() function, return the current value of 'kbshift' in bits 24..31 of D0.L.
	zero:	leave bits 24..31 alone...

themd (long) $48e
Filled in by the BIOS on a 'getmpb' call; indicates to GEMDOS the limits of the TPA. This is used by GEMDOS and should not be used by other programs.

savptr (long) $4a2
Pointer to register save area for BIOS functions.

_nflops (word) $4a6
Number of floppy disks actually attached to the system (0, 1, or 2).

sav_context (long) $4ae
Pointer to saved processor context (more on this later).

_bufl (long) $4b2
Two (GEMDOS) buffer-list headers. The first list buffers data sectors, the second list buffers FAT and directory sectors. Each of these pointers points to a BCB (Buffer Control Block), that looks like:

```
struct BCB
{
        BCB     *b_link;        /* next BCB */
        int     b_bufdrv;       /* drive#, or -1 */
        int     b_buftyp;       /* buffer type */
        int     b_bufrec;       /* record# cached here */
        int     b_dirty;        /* dirty flag */
        DMD     *b_dm;          /* -> Drive Media Descriptor */
        char    *b_bufr;        /* -> buffer itself */
} ;
```

_hz_200 (long) $4ba
Raw 200hz system timer tick. Used to divide-by-four for a 50hz system timer.

the_env (byte[4]) $4be
The default environment string.

_drvbits (long) $4c4
32-bit vector, returned by the "DRIVEMAP" BIOS function (#10), of "live" block devices. If any floppies are attached, this value is at least 3.

_dskbufp (long) $4c6
Points to a 1024-byte disk buffer somewhere in the system's BSS. The buffer is ALSO used for some GSX graphics operations, and should not be used by interrupt routines.

_prt_cnt (word) $4ee
Initialized to -1. Pressing the ALT-HELP key increments this. The screen dump code checks for $0000 to start imaging the screen to the printer, and checks for nonzero to abort the screen print.

_sysbase (long) $4f2
Points to the OS header block (in ROM or RAM).

_shell_p (long) $4f6
Points to shell-specific context.

end_os (long) $4fa
Points just past the last byte of low RAM used by the operating system. This is used as the start of the TPA (end_os is copied into _membot).

exec_os (long) $4fe
This points to the shell that gets exec'd by the BIOS after system initialization is complete. Normally this points to the first byte of the AES' text segment.

System Variables present as of Mega TOS (1.2)

****

scr_dump $502 (long)
Pointer to screen-dump code.

prv-lsto $506 (long)
Pointer to code for output device status for screen-dump when configured for "printer" port.

prv-lst $50a (long)
Pointer to code for character output for screen-dump when configured for "printer" port.

prv_auxo $50e (long)
Pointer to code for output device status for screen-dump when configured for "serial" port.

prv_aux $512 (long)
Pointer to code for character output for screen-dump when configured for "serial" port.

pun_ptr $516 (long)
Pointer to a hard-disk driver data structure; see the hard-disk driver documentation for details.

memval3 $51a (long)
Still another memory-validation marker used to check for cold boots.

System Variables present as of Mega TOS (1.2) (continued)
****

Starting at $51e, there are four sets of 8 vectors for character device functions, as follows:

```
    xconstat ds.l  8   ; $51e  console status vectors
    xconin   ds.l  8   ; $53e  console input vectors
    xcostat  ds.l  8   ; $55e  console output-status vectors
    xconout  ds.l  8   ; $57e  console output vectors
```

These allow you to manipulate character based device functions at BIOS level by replacing the built-in input/output and status routines with your own before GEMDOS gets them.

GEMDOS gets all its character input by trapping into the BIOS to a RAM based jump table.

Each set of vectors consists of the addresses of the routines that handle the BIOS character devices (in the following order:)

```
     0 - lst: (printer)
     1 - aux: (rs232)
     2 - con: (screen)
    *3 - midi
    *4 - keyboard (output only)
     5 - raw console output (bypass vt52 pressure cooker)
```

*Note: for xcostat device 3 is keyboard and 4 is midi.

No range checking is performed. If a bogus device number is passed to the BIOS' character I/O handler, the system will crash or become funky dueux.

System Variables newer than Mega TOS (1.2)
but useful retroactively
****

_longframe $59e (word)
When nonzero, indicates the presence of a CPU with long exception stack frames (i.e. not a 68000). When zero, indicates a 68000. Initialized to zero in old TOSes which are not 680x0-aware. We do *not* guarantee that newer TOSes will actually be able to accomodate other CPUs.

_p_cookies $5a0 (long)
Pointer to the "cookie jar" or zero (when there is no cookie jar). Initialized to zero (at cold boot only) by TOSes which do not install a cookie jar at boot time. See the cookie jar documentation for more details.

POST MORTEM INFORMATION

If a diagnostic cartridge is not inserted, all "unused" interrupt vectors are pointed to a handler in the BIOS that saves the processor's state in low memory (see below) and displays a number of icons in the middle of the screen. The handler attempts to restart the system after the crash -- it is not always (honestly: it isn't very often) successful.

The exact number of icons represents the exception that occurred (2 for bus error, 3 for address error, and so on -- see the `Exception Processing' section in the Motorola 68000 manual).

The processor state is saved in an area of memory that is NOT touched by a system reset. Therefore it is possible to examine a post-mortem dump after resetting the system to reboot.

```
*+
*  Post-mortem dump area;
*  processor state saved here on uncaught exception:
*
*-
proc_lives      equ     $380      ; $12345678 iff valid
proc_dregs      equ     $384      ; saved D0-D7
proc_aregs      equ     $3a4      ; saved A0-A6, supervisor A7 (SSP)
proc_enum       equ     $3c4      ; first byte is exception #
proc_usp        equ     $3c8      ; saved user A7
proc_stk        equ     $3cc      ; sixteen words popped from SSP
```

If the longword at $380 is the magic number $12345678, then the following information is valid (unless it's been stepped on by another crash).

D0-D7, A0-A6, and the supervisor A7 are copied to locations $384 to $3c0. The exception number (2 for bus error, etc.) is recorded in the byte at $3c4. The user A7 is copied to $3c8. The first sixteen words at the supervisor A7 are copied to the sixteen words starting at $3cc.

Getting Into and Out Of Supervisor Mode in GEMDOS

DRI hasn't bothered to document this function yet, so ....

Yes, there IS a way to get into (or out of) supervisor mode in GEMDOS. While you read the following description, please bear in mind that the original intent was to provide a binding usable at the C level. It is clumsy to use from assembly language.

The function is Trap 1, number 32 (hex $20). It wears three hats:

```
    LONG _super(stack)
    LONG stack;
```

If 'stack' is 1, then the function returns (in D0.L) either a 0 (indicating that the processor is in user mode) or a -1 ($FFFFFFFF indicating that the processor is in supervisor mode).

If the function is called when the processor is in user mode, GEMDOS will return with the processor in supervisor mode. The old value of the supervisor stack will be returned in D0.L. If 'stack' was NULL ($00000000), then the supervisor stack will be the same as the user stack before the call. Otherwise the supervisor stack will be set to 'stack'.

If the function is called when the processor is in supervisor mode, GEMDOS will return with the processor in user mode. 'stack' should be the value of the supervisor stack that was returned by the first call to the function.

The old value of the supervisor stack MUST restored before the process terminates. (Failure to do so may result in a crash).

An example of how to use it from C:

```
superstuff()
{
    long save_ssp;
    long trap1();

    /*
     * Get into supervisor mode:
     */
    save_ssp = trap1(0x20, 0L);

    ... do lots of supervisor stuff ....

    /*
     * Get out of supervisor mode,
     * restore old supervisor stack:
     */
    trap1(0x20, save_ssp);
}
```

And from assembly:

```
*+
*  superstuff - play around in supervisor mode
*
*-
superstuff:
        .
        .
        .  do user stuff
        .

        clr.l    -(sp)                    ; we want our own stack
        move.w   #$20,-(sp)               ; get/set supervisor mode
        trap     #1                       ; (do it)
        addq     #6,sp                    ; (clean up)
        move.l   d0,save_ssp              ; save old SSP

        .
        .
        .  do supervisor stuff
        .
        .

        move.l   save_ssp,-(sp)           ; push old SSP
        move.w   #$20,-(sp)               ; get/set supervisor mode
        trap     #1                       ; (do it)
        addq     #6,sp                    ; (clean up)

        .
        .
        .  do user stuff
        .
```

GEMDOS Relocation Format
(Clarification to GEMDOS manual)

This is the REAL GEMDOS fixup bytestream format, as implemented by the function xpgmld() in GEMDOS (as opposed to what is documented in the GEMDOS manual):

$00	no more relocation information
$01	add $FE to the dot
$02..$FF	add N to the dot, and fixup the longword there

So, to fixup a longword $100 bytes from the current one (the dot), RELMOD would generate:

```
$01 $02
```

[note that only longwords can be fixed up, and that they must be on word boundaries.]

Error Handling

Error numbers are returned by certain BIOS and most GEMDOS functions. Note that some GEMDOS functions return WORD error numbers instead of LONG ones (that is, bits 16..31 of D0.L are garbage). Someday DRI will get around to fixing these ....

[Describe critical-error handler calling conventions, whenever DRI gets around to defining them so they're useful.]

0 (OK)
Successful action (the anti-error).

-1 (ERROR)
All-purpose error.

-2 (DRIVE_NOT_READY)
Device was not ready, or was not attached, or has been busy for a long time.

-3 (UNKNOWN_CMD)
Device didn't know about a command.

-4 (CRC_ERROR)
Soft error while reading a sector.

-5 (BAD_REQUEST)
Device couldn't handle a command (the command might be valid in other contexts). Command parameters may be bad.

-6 (SEEK_ERROR)
Drive couldn't seek.

-7 (UNKNOWN_MEDIA)
Attempt to read foriegn media (usually means a corrupted or zero boot sector).

-8 (SECTOR_NOT_FOUND)
Sector could not be located.

-9 (NO_PAPER)
Printer is out of paper (this cannot happen on disks, right?)

-10 (WRITE_FAULT)
Failure on a write operation.

-11 (READ_FAULT)
Failure on a read operation.

-12 (GENERAL_MISHAP)
Reserved for future catastrophes. [This seems to be a useless error right now.]

-13 (WRITE_PROTECT)
Attempt to write on write-protected or write-only media.

-14 (MEDIA_CHANGE)
Media changed since last write -- the operation (read or write) did NOT take place. (This is more a message to the file system than a real error).

-15 (UNKNOWN_DEVICE)
Operation specified a device the BIOS doesn't know anything about.

-16 (BAD_SECTORS)
Format operation succeeded (for the most part) but yielded bad sectors.

-17 (INSERT_DISK)
Ask user to insert a disk (this is more a message to the shell -- GEM or COMMAND.PRG -- to start a dialouge with the user).

Cartridge Support

There are two kinds of cartridges. 'Application' cartridges are recognized by GEM and the desktop. 'Diagnostic' cartridges are executed almost immediately after system reset (before the 68000 touches any RAM), and may take over the entire system.

The ST hardware maps cartridge space to a 128K region starting at $FA0000, extending to $FBFFFF. The longword at $FA0000 has special meaning to the OS. It should be one of the following:

```
        $FA52255F indicates that a  diagnostic  cartridge
            is inserted.
        $ABCDEF42 indicates that an application cartridge
            is inserted.
        anything else is ignored.
```

On system RESET, if a diagnostic cartridge is inserted the OS will (almost immediately) jump to location $FA0004. A6 will contain a return address (should the cartridge ever wish to continue with system initialization). The stack pointer will be garbage. Most of the ST's hardware registers will not have been touched. The most significant of these registers is the memory controller -- the diagnostic cartridge is responsible for sizing memory and initializing the memory controller.

Application cartridges should provide 'application header' at location $FA0004 (immediately following the magic longword). An application header contains information about an application in ROM. There may be any number of applications in a cartridge.

```
CARTRIDGE APPLICATION HEADER
+------------------------+
|        CA_NEXT         | 0    ->next header
|                        |
+------------------------+
|        CA_INIT         | 4    ->init code
|                        |
+------------------------+
|        CA_RUN          | 8    ->run code
|                        |
+------------------------+
|        CA_TIME         | $c   DOS time
+------------------------+
        |       CA_DATE         | $e  DOS date
        +-----------------------+
        |       CA_SIZE         | $10 "size" of appl.
        |                       |
        +-----------------------+
        |       CA_NAME         | $14 asciz name
        |                       |     (NNNNNNNN.EEE\0)
        |                       |
        |                       |
        +-----------------------+
```

CA_NEXT is a pointer to the next application header. If CA_NEXT is $00000000, then there are no more headers in the list.

CA_INIT is a pointer to the application's initialization code. If CA_INIT is NULL, there is no initialization code. The initialization vector is called at system startup time, as controlled by magic bits in the high byte of this longword (see below).

CA_RUN is a pointer to the application's main entry point.

CA_TIME and CA_DATE are DOS-format time and date stamps. [They are kind of useful for keeping track of version numbers and things like that, but are otherwise useless ....]

CA_SIZE is a silly field that is the "size" of the application. [This field is pointless, but DRI wanted it, sooo ....]

CA_NAME is the NULL-terminate name of the application. It should be in the same format as a DOS acceptable filename, without a path (i.e. up to eight leading characters, optionally followed by a dot and up to three characters of extension, and a final NUL ($00)).

The high 8 bits (24..31) of CA_INIT have special meaning:

0 - Set to execute application (through CA_INIT vector) before interrupt vectors, display memory (etc.) have been initialized.

1 - Set to execute application (through CA_INIT vector) just before GEMDOS is initialized.

2 - (unused)

3 - Set to execute application (through CA_INIT vector) immediately before a disk-boot. [***FOR NOW*** Applicable to boot ROM only.]

4 - (unused)

5 - Set if the application is a desk accessory.

6 - Set if the application is NOT a GEM application. That is, it runs under DOS and doesn't do any AES calls.

7 - Set if non-GEM application (see bit 6) requires commandline parameters before execution.

Vertical Blank Interrupts

This section describes the OS's Vertical Blank Interrupt (VBI) handler, entered through the VBI vector at $70.

The VBI handler increments the "frame counter" 'frclock' and then checks for mutual exclusion by testing 'vblsem'. If 'vblsem' is less than or equal to zero, no other VBI code is executed. Otherwise, all registers are saved on the stack and the "vblank counter" 'vbclock' is incremented.

If the system is currently in high-resolution mode (SHIFTMD >= 2) and a low-resolution monitor is attached, the resolution is set to 'defshiftmd'. (or zero, if 'defshiftmd' is >= 2). This test is necessary because some low-resolution monitors may "burn up" when driven by the ST's high-resolution video signal.

The handler calls the cursor-blink routine.

If 'colorptr' is nonzero, then the 16 color palettes are loaded from the 16 words that 'colorptr' points to. 'colorptr' is then zeroed.

If 'screenpt' is nonzero, then the screen's physical base address set to 'screenpt'. 'screenpt' is then zeroed.

There may be any number of "deferred" VBI vectors. These are executed just before the VBI handler returns. The variable 'nvbls' contains the current number of deferred vector slots. 'vblqueue' points to an array of NVBL pointer slots that in turn point to deferred VBI code or NULL (in the case of an empty slot):

```
           +----------+
           |vblqueue o|----+
           +----------+    |
                           |
+--------------------------+
|
|   ........... 'NVBL' entries ..................
|  /                                               (
|  +-------+-------+-------+-------+-------+-------+-------+
+->|   o   |       |       |   o   |       |       |       |
   +---|---+-------+-------+---|---+-------+-------+-------+
       |                       |
       +---> handler...        +---> handler...
```

The OS initially allocates 8 VBI slots. The first slot is reserved for GEM's VBI code. To add another deferred handler, place a pointer in a free (NULL) slot. If there are no more free slots, then allocate a larger VBI array, copy the current vectors to the new array (clearing any new, unused entries), and update 'vblqueue' and 'nvbls'.

Deferred VBI handlers should return with RTS, _not_ RTE. They may use any registers except the user stack-pointer.

Applications are responsible for cleaning up vbl-vectors they have installed prior to process termination.

ROM System Initialization

[1] Initial PC set from location $FC0000, initial SP (trash, really) set from location $FC0004.

Catch system RESET. Raise processor IPL to 7, execute RESET instruction to reset hardware registers.

If a diagnostic cartridge is inserted, load a return address into A6 and jump to the cartridge.

[2] If memory was setup (i.e. this is a warmstart) the initialize the memory controller.

[3] If the RESET-bailout vector is valid, load a return address into A6 and jump to the reset handler.

[4] Initialize the PSG (deselect floppies), setup the scan rate (50 or 60hz), write default values to the color palettes, and set the display pointer to 0x10000.

If memory was sized on a previous reset, go to step 8.

[5] Size both banks of memory.

[6] [This used to perform a memory test.]

[7] Once memory has been sized and zeroed, record the fact by setting two magic longwords in low memory.

[8] Clear the low 64K of memory, from 'endosbss' to 0xffff. Initialize all kinds of OS variables. Setup interrupt vectors. Call the serial BIOS' initialization entry-point.

[9] Execute %%2 cartridge applications.

Initialize the screen resolution.

[11] Execute %%0 cartridge applications.

[12] Enable interrupts (all but HBLANK) by bringing the IPL to 3.

[13] Execute %%1 cartridge applications.

[14] Call GEMDOS' initialization routine.

[15] Attempt to boot from floppy disk, if the system variable 'bootdev' is less than 2. If there are no floppies, no attempt is made to boot from floppy.

Attempt to load a boot sector from the DMA bus. For each of the eight DMA bus devices, a read operation is attempted on logical sector 0. If the read is successful, and the sector checksums to $1234, then the sector is executed. [See the section "DMA Bus Boot"]

ALL devices are checked. The boot sector code may return, in which case the BIOS will attempt to load boot sectors from the rest of the devices.

[16] Turn on the cursor. Do autoexec. Attempt to exec COMMAND.PRG.

[17] Do autoexec. Kludge up an environment string. Exec the AES (in ROM).

If [16] or [17] ever complete, restart the system by going back to [1].

```
                    System
                    RESET
                      |
                      V
             +------------------------+
             |      Diagnostic        |  (1)
             |   Cartridge check      |
             +------------------------+
                      |
                      V
             +------------------------+
             |   Memory Controller    |  (2)
             |     (fast init)        |
             +------------------------+
                      |
                      V
             +------------------------+
             |    RESET bailout       |  (3)
             |       vector           |
             +------------------------+
                      |
                      V
             +------------------------+
             |    init PSG            |  (4)
             |    init 50hz/60hz      |
             |    init palettes       |
             |    display at $10000   |
             +------------------------+
                      |
                      V
               ~~~~~~~~~~~~~~~~~~~~~~
              ~                      ~
       YES   ~   Has memory been      ~
         /-<_    sized?  [is this      _~
         |   ~   a warmstart?]        ~
         |    ~                      ~
         |     ~~~~~~~~~~~~~~~~~~~~~~
         |                 |
         |             NO  |
         |                 |
         |                 V
         |   +------------------------+
         |   |    size memory         |
         |   |    and clear it        |
         |   +------------------------+
         |                 |
         |                 V
         |   +------------------------+
         |   |  indicate successful   |  (7)
         |   |      warmstart         |
         |   +------------------------+
         |                 |
         |                 |
```

```
     \-------------\
                   |
                   V
        +------------------------+
        |    Clear bottom 64K    | (8)
        |    Init variables      |
        |    Init interrupts     |
        |    Init serial BIOS    |
        +------------------------+
                   |
                   V
        +------------------------+
        | Execute %%2 cartridge  | (9)
        |      applications      |
        +------------------------+
                   |
                   V
        +------------------------+
        |     Init screen        | (10)
        |      resolution        |
        +------------------------+
                   |
                   V
        +------------------------+
        | Execute %%0 cartridge  | (11)
        |      applications      |
        +------------------------+
                   |
                   V
        +------------------------+
        |   Bring IPL to 3       | (12)
        |                        |
        +------------------------+
                   |
                   V
        +------------------------+
        | Execute %%1 cartridge  | (13)
        |      applications      |
        +------------------------+
                   |
                   V
        +------------------------+
        |   Initialize GEMDOS    | (14)
        |                        |
        +------------------------+
                   |
                   V
        +------------------------+ (15)
        |   Attempt to boot      |>------\
        |    from floppy         |<--\   | execute
        +------------------------+   |   | boot
                   |                 |   | sector
                   |                 \---/
```

```
                      |
                      V
           +-----------------------+ (15a)
           |   Poll devices on     |>------\
           |   DMA bus, requesting |       |
           |     boot sectors      |<--\   |  execute
           +-----------------------+   |   |  boot
                      |                |   |  sector
                      V                \---/
             ~~~~~~~~~~~~~~~~~~~~~~
            ~                      ~
      YES  ~                        ~
        /-<_   _cmdload == 0 ?       ~
        |  ~                        ~
        |   ~                      ~
        |    ~~~~~~~~~~~~~~~~~~~~~~
        |
        |             |
        |          NO |
        |             V
        |  +-----------------------+
        |  |   Turn on cursor      | (16)
        |  |   Exec \AUTO\*.PRG    |
        |  |   Exec COMMAND.PRG    |
        |  +-----------------------+
        |             |
        |             \--------------------\
        |                                  |
        \-------------\                    |
                      |                    |
                      V                    |
           +-----------------------+       |
           |   Exec \AUTO\*.PRG    | (17)  |
           |   Kludge up enviro.   |       |
           |       string          |       |
           |   Exec AES (in ROM)   |       |
           +-----------------------+       |
                      |                    |
                      |<-------------------/
                      |
                      V
                Reset system,
              start over again
```

PUNTAES and the OS Header (Gory Details)

The OS variable _sysbase [$4F2] points to the base of the operating system. The operating system may be in ROM or RAM (if _sysbase is greater than phystop then the OS is in ROM).

The base of the OS is a structure that looks like:

```
    (_sysbase) -----\
                    |
   /------------/
   |
   |    +------------------------+
   \--> | BRA to reset handler   | 0.w
        +------------------------+
        | OS version number      | 2.w
        +------------------------+
        | -> reset handler       | 4.L
        |                        |
        +------------------------+
        | -> base of OS          | 8.L
        |                        |
        +------------------------+
        | -> end of OS RAM usage | $c.L
        |                        |
        +------------------------+
        | (unused, reserved)     | $10.L
        |                        |
        +------------------------+
        | -> GEM memory usage    | $14.L
        |    parameter block     |
        +------------------------+
        | Date of system build   | $18.L
        | ($YYYYMMDD)            |
        +------------------------+
        | OS configuration bits  | $1c.w
        +------------------------+
        | DOS-formatted date the | $1e.L
        | system was built       |
        +------------------------+
        | -> GEMDOS internal     | $20.L
        |    memory        (1.2) |
        +------------------------+
        | -> Keyboard shift      | $24.L
        |    state bits    (1.2) |
        +------------------------+
        | -> Currently executing | $28.L
        |   GEMDOS process (1.2) |
        +------------------------+
```

The GEM memory usage parameter block (hereinafter referred to as ``the magic'') informs the OS about GEM's memory requirements, and GEM's start address. The magic looks like:

```
+-----------------------+
|       $87654321       |  0.L
| (our favorite magic#) |
+-----------------------+
| -> end of system      |  4.L
|    (OS+GEM) BSS       |
+-----------------------+
| -> start (execution)  |  8.L
|    address of GEM     |
+-----------------------+
                           $C
```

The OS header contains a pointer to the magic. The magic parameter block is validated if the number $87654321 appears in its first longword. GEM is started up ONLY if there is a valid magic. In addition, on a RAM-loaded system, if the magic is not valid then the memory normally used by GEM is included in the initial TPA.

The extended BIOS call puntaes() (#39) checks to see if the magic is valid. If the magic is NOT valid, it returns immediately. Otherwise it checks if the magic is located in ROM, and if it is, puntaes() returns. Finally puntaes() invalidates the magic (by zeroing its first longword) and jumps to the system reset handler.

Puntaes will either return (meaning that the AES was already punted, or more accurately, that the magic was invalid) or clobber the magic and restart the operating system. The OS must be restarted because GEMDOS does not allow the TPA to be expanded after GEMDOS has been initialized [fooey!].

The country-specific configuration word (``os_conf'') looks something like:

```
                2       1       0
-  -  --+-------+-------+-------+
        |               |       |
        |   country#    | PAL/  |
        |               |  NTSC |
-  -  --+-------+-------+-------+
```

The country-number assignments are:

0	USA
1	Germany
2	France
3	UK
4	Spain
6	Sweden
7	Switzerland (French)
8	Switzerland (German)
9	Turkey

Bit 0 of the word indicates NTSC when 0 and PAL when 1; the ``syncmode'' hardware register is initialized accordingly during system startup. The country bits may be expanded in the future.

The version number is $0000 for the boot ROM, and nonzero for ROM-based operating systems. The format of the version word is $VVRR (VV = version#, RR = release#), and the first OS ROMs will have the version $0100. Mega ROMS (blitter support) have the version number $0102.

Several dates, in various formats, are in the header. The first is (more or less) human-readable, in hexadecimal it is a longword that reads like $YYYYMMDD (YYYY = year, MM = month, DD = day). The second date is a GEMDOS-format timestamp.

At an offset $20 from the address at _sysbase is a pointer, _root, which holds the base of the OS pool, the internal memory used by GEMDOS. This pointer is used by FOLDERXXX.PRG. You can still add to the pool the same way as before, but the OS will take the memory you added and use it differently than before. Exists since Mega ROMS.

A pointer to the variable kbshift is at an offset of $24 from _sysbase. This is a word which contains the keyboard shift state bits which is updated at interrupt level. Exists since Mega ROMS.

The process ID (basepage address) of the process GEMDOS is currently executing is held by the variable _run (long) and is at an ofset of $28 from _sysbase. Exists since Mega ROMS.

DISCLAIMER
Atari makes no promises that version numbers in future revisions of the operating system will reflect reality, since the outside world's version of reality is different from Atari's. We may release bug fixes without changing the OS version number, or (contrariwise) we may change version numbers without changing the operating system.

Boot Sectors

The boot sector contains

- o A volume serial number
- o A BIOS parameter block
- o Optional boot code and boot parameters

An executable boot sector must word-checksum to the magic number $1234. During system initialization the boot sector from a disk drive is loaded into a buffer. If the checksum is correct, the system JSRs the first byte of the buffer. [Since the location of the buffer is indeterminant, any code contained in the boot sector must be position-independent.] See the section on system initialization for further details on writing bootable applications.

When a "Get BPB" call is made, the BIOS reads the boot sector and examines the prototype BIOS parameter block (BPB). A BPB is constructed from the prototype. If the prototype looks strange (for instance, if critical fields in it are zero) the BIOS returns NULL (as an error indication).

A BPB is normally computed and written when the volume is formatted.

The 24-bit serial number is used to determine if the user has changed disks. (see the [still nonexistant] section on "Disk Changes"). The serial number is computed and written by the FORMAT utility, and is (hopefully) unique.

```
+------------------------+
|       BRA.S            | $0   branch to boot code
|       (wherever)       |
+------------------------+
|       filler           | $2   reserved for OEMs
|                        |
|       (OEM             |
|        cruft)          |
|                        |
|                        |.
+------------------------+
|       SERIAL           | $8   volume serial number
|       24-bit volume    |      written by FORMAT
|       serial number    |
+------------------------+
|l      BPS              | $b   #bytes/sector
|h                       |
+------------------------+
|       SPC              | $d   #sectors/cluster
+------------------------+
|l      RES              | $e   #reserved sectors
|h                       |
+------------------------+
|       NFATS            | $10  #FATs
+------------------------+
|l      NDIRS            | $11  #directory entries
|h                       |
+------------------------+
|l      NSECTS           | $13  #sectors on media
|h                       |
+------------------------+
|       MEDIA            | $15  media descriptor
+------------------------+
|l      SPF              | $16  #sectors/FAT
|h                       |
+------------------------+
|l      SPT              | $18  #sectors/track
|h                       |
+------------------------+
|l      NSIDES           | $1a  #sides on media
|h                       |
+------------------------+
|l      NHID             | $1c  #hidden sectors
|h                       |
+------------------------+
|       boot code        | $1e
.       (if any)         .

|                        |
+------------------------+
                           $200
```

The prototype BPB is software compatible with an MS-DOS version 2.x BPB. (This does not mean the ST can read sectors written by, or write sectors readable by, a disk controller other than the WDC 1770/1772).

The low byte of a 16-bit field in the BPB (such as 'BPS') occupies the lower address [as on the 8086.]

BPS is the number of bytes per sector (for floppies on the ST, it will be 512).

SPC is the number of sectors per cluster (on floppies, usually 2 for a cluster size of 1K).

RES is the number of reserved sectors at the beginning of the media, including the boot sector. RES is usually 1 on floppies.

NFATS is the number of File Allocation Tables on the media.

NDIRS is the number of directory entries.

NSECTS is the total number of sectors on the media (including the reserved sectors).

MEDIA is a media descriptor byte. The ST BIOS does not use this byte, but other file-systems might.

SPF is the number of sectors in each FAT.

SPT is the number of sectors per track.

NSIDES is the number of sides on the media. (Single-sided media can be read on double-sided drives, but not vice-versa).

NHID is the number of "hidden" sectors. (The ST BIOS currently ignores this value for floppies).

The last word in the boot sector (at offset $1FE) is reserved for "evening out" checksums. In particular, the "_protobpb" extended BIOS function modifies this word.

Formatting a Floppy Disk

[1] Use the 'flopfmt()' (#10.) extended BIOS call to format all tracks on the floppy disk. If tracks 0 or 1 have any bad sectors then the media is unusable.

The ST standard format is

1 or 2 sides;
80 tracks;
9 sectors per track;
no interleave (sequential sectors).

Zero the first two tracks (this will zero the FAT and directory sectors).

[2] Use the 'protobt()' (#18.) extended BIOS call to create a boot sector. The 'disktype' parameter should be 2 or 3 for 1 or 2 sided 80-track media respectively. The 'serialno' parameter should be a random number (or $1000000).

The 'execflag' parameter should be zero unless the prototyping buffer contains code (such as a copy of the Loader) that you want executed when the disk is booted.

[3] Write the boot sector, (prototyped in the buffer in step [2]) to track 0, side 0, sector 1 of the new disk. Do NOT use the 'rwabs' call; use the extended BIOS function 'flopwr'.

It is possible to create disks in weird formats by varying the number of sectors per track, formatting a few extra tracks, or specifying strange interleave factors.

The 1772 "write track" codes used to format a track are:

```
COUNT   BYTE    what
-----   ----    ----------------
60      $4e     (start of track)

For each sector:
 12     $00
 3      $f5     (writes $a1)
```

1	$fe	(ID address mark)
1	track#	(0..$4f)
1	side#	(0..1)
1	sector#	(1..9)
1	$02	(512 bytes/sector)
1	$f7	(2 CRCs written)
22	$4e	
12	$00	
3	$f5	(writes $a1)
1	$fb	(data address mark)
512	xx	(virgin data)
1	$f7	(2 CRCs written)
40	$4e	

End of track:

1401	$4e	(filler at end of track)

DMA Bus Boot Code

This code, extracted from the ST's BIOS, attempts to load boot sectors from devices on the DMA bus. The code can be used:

- o As an example of how to use the DMA bus (useful for boot-sector and device-driver writers);
- o To provide information about the timeout and command characteristics expected from bootable DMA bus devices;

```
gpip            equ     $fffffa01       ; (B) 68901 input register

diskctl         equ     $ffff8604       ; (W) disk controller data access
fifo            equ     $ffff8606       ; (W) DMA mode control
dmahigh         equ     $ffff8609       ; (B) DMA base high
dmamid          equ     $ffff860b       ; (B) DMA base medium
dmalow          equ     $ffff860d       ; (B) DMA base low

flock           equ     $43e            ; (W) DMA chip lock variable
_dskbufp        equ     $4c6            ; (L) -> 1K disk buffer
_hz_200         equ     $4ba            ; (L) 200hz counter

*+
*  dmaboot - attempt to boot from a device on the DMA bus
*    Passed:    nothing
*
*    Returns:   maybe-never (although it depends ...)
*
*    Uses:      everything
*
*    Discussion:
*               Attempts to read boot sectors from eight devices connectec
*               to the DMA bus.  If a sector is read, and it is executable
*               (word checksum is $1234), then it is executed.
*
*               This code should take about 0.5 sec to execute if nothing
*               is connected to the DMA bus.  Of course, if something IS
*               hooked up, it should provide us with a boot sector, right
*
*-
dmaboot:
        moveq   #0,d7           ; start with dev #0
```

```
dmb_1:  bsr     dmaread             ; attempt to read boot sector
        bne     dmb_2               ; (failed -- try next dev)
        move.l  _dskbufp,a0         ; a0 -> disk buffer
        move.w  #$00ff,d1           ; checksum $100 words
        moveq   #0,d0               ; checksum = 0
dmb_3:  add.w   (a0)+,d0            ; add (next) word
        dbra    d1,dmb_3
        cmp.w   #bootmagic,d0       ; is the sector executable?
        bne     dmb_2               ; (nope)
        move.l  _dskbufp,a0         ; a0 -> disk buffer
        jsr     (a0)
dmb_2:  add.b   #$20,d7             ; next devno
        bne     dmb_1               ; (do all eight devs)
        rts


*+
*  dmaread - attempt to read boot sector from DMA bus device
*    Passed:     d7.b = ddd00000
*                ('ddd' is the ACSI device number, 0..7)
*
*    Returns:    NE: read failed;
*                EQ: successful read,
*                    sector data in (*_dskbufp)[];
*
*    Preserves: d7.w
*
*    Uses:       everything else
*
*
*-
dmaread:
        lea     fifo,a6             ; a6 -> DMA control register
        lea     diskctl,a5          ; a5 -> DMA data register
        st      flock               ; lock up DMA against vblank

        move.l  _dskbufp,-(sp)      ; setup DMA pointer
        move.b  3(sp),dmalow
        move.b  2(sp),dmamid
        move.b  1(sp),dmahigh
        addq    #4,sp

        move.w  #$098,(a6)          ; toggle R/W, leave in Read state
        move.w  #$198,(a6)
        move.w  #$098,(a6)
        move.w  #1,(a5)             ; write sector count register (= 1)

        move.w  #$088,(a6)          ; select dma bus (not SCR)

        move.b  d7,d0               ; setup d0.L with devno+command
        or.b    #$08,d0             ; d0.b = devno<<5 .OR. "READ" command bits
        swap    d0
        move.w  #$088,d0
        bsr     wcbyte             ; d0.L = xxxxxxxxDDD01000xxxxxxx010001010
        bne     dmr_q              ; (punt on timeout)

        moveq   #3,d6              ; (count = 4)
        move.l  #$0000008a,d0      ; d0.L = generic command ($0000)
dmr_lp: bsr     wcbyte             ; write bytes 2, 3, 4 and 5
        bne     dmr_q              ; (punt on timeout)
        dbra    d6,dmr_lp          ; (loop for more bytes)

        move.l  #$0000000a,(a5)    ; write byte 6 (final byte)
        move.w  #400,d1            ; timeout = 2.0 sec
        bsr     wwait              ; wait for completion
        bne     dmr_q              ; (punt on timeout)

        move.w  #$08a,(a6)         ; select status reg
        move.w  (a5),d0            ; get return code from DMA device
        and.w   #$00ff,d0          ; strip crufty bits
        beq     dmr_r              ; (return if OK)

*--- reset DMA, return NE
dmr_q:
        moveq   #-1,d0             ; return -1 (error)
dmr_r:  move.w  #$080,(a6)         ; cleanup DMA chip for floppy driver
        tst.b   d0                 ; (test for NE on return)
        sf      flock              ; unlock DMA chip
        rts                        ; return


*+
*  wcbyte - write ACSI command byte, wait for IRQ
*    Passed:    D0.L = command byte and FIFO control
*                       bits 16..23 = command byte,
*                       bits 0..7 = FIFO control bits
*               a5 -> $ff8604
*
*    Returns:   NE on failure (timeout)
*               EQ on successful ACK
*
*    Uses:      d1
*
*-
wcbyte:
        move.l  d0,(a5)            ; write WDC, WDL [due to jwt]
        moveq   #10,d1             ; wait 1/20th second
wwait:  add.l   _hz_200,d1         ; d1 = time to quit at...
ww_1:   btst.b  #5,gpip            ; disk done?
        beq     ww_w               ; (yes, return)
        cmp.l   _hz_200,d1         ; timeout?
        bne     ww_1               ; (not yet -- wait some more...)
        moveq   #-1,d1             ; ensure NE (timeout error) return
ww_w:   rts
```

The Loader

The Loader is a generic system-loader. It lives on boot sectors, and is brought into RAM and executed during system initialization. The Loader has the capability to load an "image" file or a set of contiguous sectors from disk.

The six reserved bytes starting at offset 2 in the boot sector must be:

'Loader'

for some tools to be able to manipulate Loader boot sectors.

An image file contains no header or relocation information. It is an exact image of the program to be executed. The loader is capable of loading any file from disk, regardless of where it appears in the directory or whether the file is contiguous or not.

Loader information immediately follows the BPB in the boot sector:

```
+-----------------------+
|        EXECFLG        | $1e
|                       | _cmdload
+-----------------------+
|        LDMODE         | $20
|                       | load mode
+-----------------------+
|        SSECT          | $22
|                       | sector start
+-----------------------+
|        SECTCNT        | $24
|                       | #sectors
+-----------------------+
|        LDADDR         | $26
|                       | load-address
|                       |
|                       |
+-----------------------+
|        FATBUF         | $2a
|                       | FAT address
|                       |
|                         |
+-------------------------+
|       FNAME   n         |  $2e
|               n         |
|               n         |
|               n         |
|               n         |
|               n         |
|               n         |
|               n         |
|               e         |
|               e         |
|               e         |
+-------------------------+
|       (reserved)        |  $39
+-------------------------+
|       BOOTIT            |  $3a
|       code              |
.                         .
```

EXECFLG is a word that is copied to '_cmdload'.

LDMODE governs the loading mode. If LDMODE is zero, a file is searched for and loaded. If LDMODE is nonzero, then 'SECTCNT' sectors, starting with logical sector number 'SSECT', are loaded from the disk.

SSECT is the logical sector number to start loading from (valid iff LDMODE is nonzero).

SECTCNT is the number of sectors to load (valid iff LDMODE is nonzero).

LDADDR is the load-address of the file (or the sectors).

FATBUF points to a place to put the FAT and directory sectors.

FNAME is a filename to load (valid iff LDMODE is zero). It consists of eight character name and a three character extension.

[See also: documentation on the BOOTGEN utility.]

Boot Sequence

[1] The boot sector is loaded. The Loader takes control of the system.

[2] The boot device's directory and 2nd FAT buffer are read into memory, starting at _membot. The Loader searches for a file (usually) called TOS.IMG. If it is not found, it returns with an error code in D0.

[3] TOS.IMG is read into memory, starting at $40000.

[4] Control is passed to the first byte of TOS.IMG.

TOS.IMG consists of three parts:

[1] A relocator (RELOCRL) that moves TOS.IMG to where it expects to be executed in memory. RELOCRL takes control of the system, fades the screen, performs a fast block-copy, and passes control to the first byte in the operating system.

[2] An image of the operating system ('prox 90K).

[3] An image of the desktop and GEM ('prox 110K).

System initialization proceeds as normal (except for clearing memory) once the OS has control.

Boot ROM

The ST boot ROM (AKA "Das Boot") contains a subset of the BIOS. The only functions available relate to reading floppy disks.

System initialization is identical to the normal OS procedure. However, the locations and intepretations of the system variables may have changed. See the end of this section for a list of "safe" system variables.

The normal course of events is:

> The boot ROM catches RESET and initializes the system. It puts up some pretty graphics. Kids'll love it.
>
> An attempt is made to boot from both floppies. '_bootdev' will contain the device number on a successful boot sector load. [Someday there may be a version of the boot ROM that understands about hard disks.]
>
> The boot sector is executed. [See-also: Loader]

Das Boot's version number (the second word in the ROM, at $FC0002) is $0000.

BIOS functions on trap 13:

func	Name [see: GEMDOS spec]
0:	[unused]
1:	[unused]
2:	[unused]
3:	[unused]
4:	rwabs (read only)
5:	[unused]
6:	[unused]
7:	getbpb

Extended functions on trap 14:

func	Name [see: Extended BIOS Functions]
0:	[unused]

1:	ssbrk
2:	[unused]
3:	[unused]
4:	[unused]
5:	[unused]
6:	[unused]
7:	[unused]
8:	_floprd (read sectors)

DAS BOOT uses memory from $10000 to $20000 for screen buffers. Avoid loading stuff into this region (until you take over the system) when writing directly-bootable applications.

Between the time when DAS BOOT was released and the time the first RAM-loaded systems were shipped (will be shipped?) the variables in low memory were added to and relocated.

<<<give list of "safe" variables here>>>

GEMDOS CALL
(QUICKER) REFERENCE GUIDE

Functions are available through trap #1. The first number is the trap number (first word on the stack when the trap is made). The function's name (as given in OSBIND.H) is next, along with the named arguments. The number in brackets is the number of bytes that must be cleaned up off the stack after the call is made (for those of us doing traps from assembly). The argument declarations (if any) follow the first line. Then a short description of the function is given.

In general, GEMDOS calls return LONGs in D0. However, there /are/ exceptions. When testing for error returns, it is best to examine D0.W only. In addition, GEMDOS may occasionally return BIOS error numbers (that is, between -1 and -31).

```
$00 Pterm0() [2]
          Terminate process (with return code of $0).

$01 Cconin() [2]
          Return cooked character from stdin.

$02 Cconout(chr) [4]
    char chr;
          Write character to stdout.

$03 Cauxin() [2]
          Return character from AUX:.

$04 Cauxout(chr) [4]
    char chr;
          Write character to AUX:.

$05 Cprnout(chr) [4]
    char chr;
          Write character to PRN:.

$06 Crawio(wrd) [4]
    WORD wrd;
          If (wrd == 0x00ff) return char from stdin
          If (wrd != 0x00ff) print it on stdout;
```

```
$07 Crawcin() [2]
            Return raw character from stdin (without echo).

$08 Cnecin() [2]
            Read char from  stdin  without  echo.  Control
            characters  (^S,  ^Q,  ^C)  are interpreted and
            have effect.

$09 Cconws(str) [6]
    char *str;
            Write null-terminated string to stdout.

$0a Cconrs(buf) [6]
    char *buf;
            Read  edited  string  from  stdin.  On  entry,
            buf[0] contains size of data part of buf[].  On
            exit, buf[1] contains number of  characters  in
            data  part  of  buf[].   The data part of buf[]
            starts at buf[2].

$0b Cconis() [2]
            Return -1 [nonzero] if character  is  available
            on stdin, 0 otherwise.

$0e Dsetdrv(drv) [4]
    WORD drv;
            Select  current  drive  (0=A:,  1=B:,  etc.).
            Returns a bitmap of drives in the system (bit 0
            = A, ....)

$10 Cconos() [2]
            Returns -1 [nonzero] if  console  is  ready  to
            receive a character, 0 if it is "unavailable."

$11 Cprnos() [2]
            Returns  -1  [nonzero]  if  PRN:  is  ready  to
            receive a character, 0 if it is "unavailable."

$12 Cauxis() [2]
            Returns -1 [nonzero] if char  is  available  on
            AUX:, 0 otherwise.

$13 Cauxos() [2]
            Returns  -1  [nonzero]  if  AUX:  is  ready  to
            receive a character, 0 if it is "unavailable."

$19 Dgetdrv() [2]
            Returns number of current drive (0=A:, etc.)

$1a Fsetdta(ptr) [6]
    LONG ptr;
            Set disk transfer address (used by Fsfirst()).
```

```
$20 Super(stack) [6]
    LONG stack;
           Hack processor privilege mode.  If  'stack'  is
           1L,  return  0  or  -1 (processor is in user or
           supervisor mode).  If in user mode,  switch  to
           supervisor mode and use 'stack' as the supervi-
           sor stack (or the value from USP if 'stack'  is
           NULL).   If  in supervisor mode, switch to user
           mode and use 'stack' as the  supervisor  stack.
           Return the old supervisor stack value.

$2a Tgetdate() [2]
           Returns date:

               bits
               0..4    day 1..31
               5..8    month 1..12
               9..15   year 0..119 since 1980


$2b Tsetdate(date) [4]
    WORD date;
           Set date in the format described above.

$2c Tgettime() [2]
           Return time in the format:

               bits
               0..4    second 0..59 (2-second resolution)
               5..10   minute 0..59
               11..15  hour 0..23


$2d Tsettime(time) [4]
    WORD time;
           Set time in the format described above.

$2f Fgetdta() [2]
           Return current DTA.

$30 Sversion() [2]
           Return current version number.

$31 Ptermres(keep, ret) [8]
    LONG keep;
    WORD ret;
           Terminate and stay resident.  'keep' has number
           of  bytes  to  keep  in the process descriptor.
           'ret' is the process' return code.

$36 Dfree(buf, drv) []
    LONG buf;
    WORD drv;
```

```
             Return information about  allocation  on  drive
             'drv'  (0=current,  1=A:,  2=B:,  etc.).  'buf'
             points to  a  structure  where  stuff  will  be
             returned:

                 LONG b_free;    #free clusters on drive
                 LONG b_total;   total #clusters on drive
                 LONG b_secsiz;  #bytes in a sector
                 LONG b_clsiz;   #sectors in a cluster


   $39 Dcreate(path) [6]
       char *path;
             Create a directory.

   $3a Ddelete(path) [6]
       char *path;
             Delete a directory.

   $3b Dsetpath(path) [6]
       char *path;
             Set current directory.

   $3c Fcreate(name, attr) [8]
       char *name;
       WORD attr;
             Create a file with the given pathname.  Returns
             a  handle  or a (negative) error#.  Bits in the
             attribute word are:

                 $01     set to readOnly
                 $02     hidden from directory search
                 $04     system file, hidden from dir search
                 $08     volume label (first 11 bytes of name)


   $3d Fopen(name, mode) [8]
       char *name;;
       WORD mode;
             Open a file.  Mode is  0,  1  or  2  for  read,
             write,  and  read/write.  Returns a handle or a
             (negative) error#.

   $3e Fclose(handle) [4]
       WORD handle;
             Close the handle.

   $3f Fread(handle, count, buf) [12]
       WORD handle;
       LONG count;
       char *buf;
             Read bytes from a file.  Return count read,  or
             a negative error#.
```

```
$40 Fwrite(handle, count, buf) [12]
    WORD handle;
    LONG count;
    char *buf;
          Write bytes to a file.  Return  count  written,
          or a negative error#.

$41 Fdelete(name) [6]
    char *name;
          Delete the file.

$42 Fseek(offset, handle, mode) [10]
    LONG offset;
    WORD handle;
    WORD mode;
          Seek within the file (handle).  'offset' is the
          (signed)  number  of bytes to seek by.  Mode is
          one of:

              0       from beginning of file
              1       from current position
              2       from end of file


$43 Fattrib(path, mode, mode) [10]
          Get file attributes if 'mode' is 0, set them if
          'mode' is 1.  Bits are:

              $01     readOnly
              $02     hidden
              $04     system (hidden hidden)
              $08     volume label
              $10     subdirectory
              $20     written to and closed


$45 Fdup(stdhandle) [4]
    WORD stdhandle;
          Returns non-standard handle that refers to  the
          same file.

$46 Fforce(stdhandle, nonstdhandle) [6]
    WORD stdhandle;
    WORD nonstdhandle;
          Force standard handle to point to same file  or
          dev as the nonstandard handle.

$47 Dgetpath(pathbuf, drv) [8]
    char *pathbuf;
    WORD drv;
          Return  current  directory  for  drive   'drv'
          (0=default,  1=A:, etc.) in the buffer.  Buffer
          must be at least 64 bytes long.
```

```
$48 Malloc(amount) [6]
    LONG amount;
          'amount' contains # bytes to allocate  (or  -1,
          which   returns   maximum   available  memory).
          Return pointer to block (on word  boundary)  of
          'amount' bytes, or zero on allocation failure.

$49 Mfree(addr) [6]
    char *addr;
          Free a block  of  memory.  Nonzero  return  on
          failure.

$4a Mshrink(zero, mem, size) [12]
    WORD zero;
    LONG mem;
    LONG size;
          'zero' must be a word containing 0.  'mem' con-
          tains beginning of memory block.  'size' is the
          the amount of memory to RETAIN  in  the  block.
          Nonzero return on failure.

$4b Pexec(mode, path, commandline, environment) [16]
    WORD mode;
    char *path;
    char *commandline;
    char *environment;
          'mode' is one of:

              0       load and go
              3       just load
              4       just go
              5       create basepage

          'commandline' is the  command  tail,  which  is
          copied into the basepage.  'environment' is the
          environment string; if NULL, the parent process'
          environment string is inherited.

          For mode 0, the  return  code  is  the  child's
          return  code, or a negative (OS) error.  If the
          load or create-basepage fails, a negative error
          number is returned.

$4c Pterm(code) [4]
    WORD code;
          Terminate current process, returning 'code'  to
          the parent.

$4e Fsfirst(spec, attr) [8]
    char *spec;
    WORD attr;
          'attr' is a set of  attributes  to  match  (see
          function  #43 for details).  'spec' may contain
```

wildcard characters in the filename, but not in the pathname. Returns 0 if a file is found, EFILNF if no file was found. Dumps stuff into the DTA:

bytes	
0..20	junk
21	file attributes
22-23	file time stamp
24-25	file date stamp
26-29	file size (longword)
30-43	name+extension of found file

```
$4f Fsnext() [2]
          Continue with with Fsfirst().

$56 Frename(zero, old, new) [12]
    WORD zero;
    char *old;
    char *new;
          Change the name of a file from 'old' to 'new'.
          'zero' is reserved, and must be 0.

$57 Fdatime(handle, buf, set) [10]
    WORD handle;
    char *buf;
    WORD set;
          'buf' points to buffer containing file date and
          time information. 'handle' is a handle to the
          file. If 'set' is zero, get the time and date.
          If 'set' is 1, set the file time and date.
```